Informatique industrielle A7-19571
Systèmes temps-réel

J.F.Peyre

Partie 5 : Gérer la communication et la synchronisation inter-tâches

Plan du cours

- Introduction aux problèmes de synchronisation
- Sémaphores : principe et utilisation
- Moniteurs : un concept plus évolué

Introduction

Introduction

- **Programme multi-tâches : coopération inter-tâches**
  - Echange ou partage d'informations
  - Synchronisation pour le respect de la causalité et pour la protection des données
- **Deux modèles**
  - Système centralisé : privilégie la communication et la synchronisation par mémoire commune
  - Système distribué : privilégie la communication et la synchronisation par messages

Caractéristiques des systèmes distribués

- **Ensemble de machines reliées en réseau (pouvant être temps réel)**
- **Pas de mémoire commune (et plusieurs ordonnanceurs)**
- **Coopération et synchronisation basées sur *l'échange de messages***
  - permet l'échange de données (un message transporte des données)
  - synchronisation implicite : on ne peut recevoir un message que s'il a été émis
  - permet aussi une synchronisation explicite : le message peut transporter une information de synchronisation

Mécanismes présents dans les systèmes distribués : mécanismes de communication

- **Appel de procédures distantes**
  - envoie d'une demande à un serveur
  - plusieurs sémantiques (appel bloquant ou non bloquant)
  - mécanisme système
  - exemple : RPC

- **Invocation de méthodes distantes**
  - appel d'une méthode d'un objet situé sur un site distant
  - mécanisme langage
  - exemple : Java RMI (Remote Method Invocation)

Caractéristiques des systèmes centralisés

- Une seule machine pour plusieurs tâches : *partage obligatoire*
- Mémoire commune
- Communication et synchronisation par *partage de données* en mémoire
  - simple à mettre en oeuvre
  - pose le problème de la protection des données par rapport à des accès multiples (problème de cohérence) lorsque les opérations ne sont pas atomiques

Mécanismes présents dans les systèmes centralisés : mécanismes de protection

- **Masquage des interruptions**
  - dangereux, à réserver à des zones très ciblées (noyau système)
- **Test and Set**
  - bas niveau, plutôt matériel (au niveau du processeur)
- **Sémaphore (Dijkstra - 1965)**
  - assez bas niveau mais très commun et simple dans le cas général
- **Moniteur**
  - haut-niveau, le plus commode et le plus sûr (au niveau langage)
- **Mécanismes basés sur l'échange de données également possible**

Synchronisation dans les systèmes centralisés

Pourquoi des mécanismes de protection ?

- **Considérons l'exemple suivant :**
  - Deux tâches partagent un registre
  - Le registre est en fait un tableau de N entiers
  - Les tâches doivent régulièrement incrémenter chaque case du registre
  - L'incrémentation doit se faire en N opérations élémentaires

Nous étudions un programme implémentant cet exemple en C avec la norme Posix puis avec le langage Ada

Exemple incorrecte (C/Posix) : primitive E/S

```
#include <stdio.h>
#include <pthread.h>

#define TAILLE_REGISTRE 10
pthread_t id1, id2;
```

```
typedef int ValeurRegistre[TAILLE_REGISTRE];
ValeurRegistre LeRegistre;
```

```
void LitRegistre(ValeurRegistre R){
   int i;

   for (i=0; i<TAILLE_REGISTRE; i++) R[i] = LeRegistre[i];
}
```

```
void EcritRegistre(ValeurRegistre R){
   int i;

   for (i=0; i<TAILLE_REGISTRE; i++) LeRegistre[i] = R[i];
}
```

./...

Exemple incorrecte (C/Posix) : code des tâches

```
void f(void){
    int i, j;
    ValeurRegistre RLocal;

    for(j=0; j < 1000000; j++){
        LitRegistre( RLocal );
        for(i=0; i<TAILLE_REGISTRE; i++){
                RLocal[i] ++;
        }
        EcritRegistre( RLocal );
    }
}
```

./...

Exemple incorrecte (C/Posix) : le principal

```
main(){
    int i;

    for(i=0; i<TAILLE_REGISTRE; i++)
        LeRegistre[i] = 0;

    pthread_create(&id1, NULL, (void *(*)()) f, NULL);
    pthread_create(&id2, NULL, (void *(*)()) f, NULL);

    pthread_join(id1, NULL);
    pthread_join(id2, NULL);

    printf("Valeur finale de cpt : \n");

    for(i=0; i<TAILLE_REGISTRE; i++)
        printf("%d \n", LeRegistre[i]);
}
```

Exemple incorrecte (Ada) : primitive E/S

```
with Text_IO; use Text_IO;

procedure Prog_Exc is

Taille_Registre : constant Natural := 10;

type Valeur_Registre is array(1..Taille_Registre) of Integer;

Le_Registre : Valeur_Registre;
```

```
function Lire_Registre return Valeur_Registre is
begin
   return Le_Registre;
end;
```

```
procedure Ecrire_Registre(R: in Valeur_Registre) is
begin
   Le_Registre := R;
end;
```

./...

Exemple incorrecte (Ada) : code des tâches

```
task type T_F;
```

```
task body T_F is
   R_Local : Valeur_Registre;
begin
   for I in 1..1_000_000 loop
       R_Local := Lire_Registre;

       for I in R_Local'Range loop
              R_Local(I) := R_Local(I) + 1;
       end loop;

       Ecrire_Registre( R_Local);
   end loop;

end T_F;
```

./...

Exemple incorrecte (Ada) : le principal

```
begin -- début du main

   Le_Registre := (others => 0);

   declare
       T1, T2: T_F; -- déclaration de deux tâches de type T_F
   begin
       null; -- execution de t1 et de t2 en concurrence; on attend la fin
   end;

   Put_line("Valeur finale du registre ");

   for I in Le_Registre'Range loop
       Put_line(Integer'Image( Le_Registre(I) ));
   end loop;

end Prog_Exc;
```

Exemple incorrecte : exécution du programme

Voici un exemple d'exécution (C/Posix ou Ada)

```
Valeur finale du registre
     1543280
     1543280
     1177867
     1177867
     1177867
     1177867
     1177867
     1177867
     1177867
     1177867
```

On aurait du avoir 10 lignes avec la valeur 2000000 (deux millions)

il est nécessaire de garantir qu'une séquence lecture / incrémentation / écriture soit finie avant qu'une autre ne puisse commencer !

Mécanismes de synchronisation

- **Deux mécanismes classiques peuvent être utilisés pour protéger des données accédées par plusieurs tâches :**
  - Les **sémaphores** qui ne peuvent être pris qu'un nombre déterminé de fois (si le sémaphore est déjà pris une nouvelle tentative bloquera l'appelant)
  - Les **moniteurs** qui permettent « d'encapsuler » des données en définissant des règles d'accès exclusif à ces données

Les sémaphores

- **Un sémaphore (Dijkstra 1965) est un élément de synchronisation auquel est associée trois données :**
  - un compteur interne entier (on le note S_CPT)
  - une file d'attente
  - deux opérations atomiques (exécution indivisible) nommées P et V
  - une opération atomique d'initialisation (et création) que l'on notera E0

- **Si S est un sémaphore, on notera P(S) ou V(S) ou E0(S,V) l'appel d'une opération sur le sémaphore S**

- **On dit qu'un sémaphore est binaire si son compteur reste inférieur ou égal à 1**

Les sémaphores : sémantique

P(S) : prend le sémaphore; *bloquant* si déjà pris

```
S_CPT --;
Si (S_CPT < 0) Alors
   Bloquer l'appelant et le mettre dans la file d'attente associée à S;
fin si
```

V(S) : rend le sémaphore; jamais bloquant

```
S_CPT ++;
Si (S_CPT <= 0) Alors
   Choisir un processus dans la file d'attente, le retirer de celle-ci et
   le réveiller (le débloquer);
fin si
```

EO(S, V) : initialise le compteur interne à la valeur V

```
S_CPT <-- V;
```

Les sémaphores : sémantique (suite)

- **On a l'invariant suivant *S_CPT < 0 => Abs(S_CPT) = Lg(File Attente)***

- **Dans le cas d'un sémaphore binaire, l'opération V(S) n'aura aucun effet si le compteur du sémaphore est déjà à un 1**

- **Il n'est pas forcément précisé quelle tâche est réveillée dans le cas où il y en a plusieurs en attente**
  - gestion FIFO (on réveille la plus ancienne dans la file)
  - gestion par priorité (on réveille la plus prioritaire)

- **La manipulation de sémaphores peut introduire le phénomène d'inversion de priorité étudié le cours précédent**

Les sémaphores : exemple d'utilisation

- **Dans l'exemple précédent d'exécution incorrecte il faut "encadrer" les actions de**
  - lecture du registre
  - incrémentation des cases du registres
  - écriture du registre

  par la prise et le relâchement d'un sémaphore binaire (initialisé à 1)

```
P(S);
    lecture du registre
    incrémentation des cases constituant le registre
    écriture du registre
V(S);
```

- **De la sorte, la séquence des opérations de lecture, d'incrémentation et d'écriture devient une opération atomique : il n'y a plus d'incohérence de la valeur commune du registre (une nouvelle opération ne peut commencer que lorsque celle qui est en cours est terminée)**

Les sémaphores : guide d'utilisation

- **Assurer une bonne protection (no trop ni trop peu)**
  - N'utiliser les sémaphore que sur de petites sections critiques
  - Faire en sorte que celui qui rend le sémaphore soit celui qui l'a pris
  - Isoler les actions en section critique du reste du code
- **Eviter l'interblocage en ordonnant la prise des sémaphores ou utiliser des tableaux de sémaphores à la Unix**
  - exemple d'interblocage :
    - la tâche A prend S1, la tâche B prend S2 (*S1 et S2 sont deux sémaphores binaires*)
    - la tâche A tente de prendre S2 : elle se bloque
    - la tâche B tente de prendre S1 : elle se bloque

    -> ***on est dans une situation d'interblocage***

Les moniteurs

- **Les moniteurs proposent une solution de "haut-niveau" pour la protection de données partagées (Hoare 1974)**
- **Ils simplifient la mise en place de sections critiques**
- **Ils sont définis par**
  - des données internes (appelées aussi variables d'état)
  - des primitives d'accès aux moniteurs (points d'entrée)
  - des primitives internes (uniquement accessibles depuis l'intérieur du moniteur)
  - une ou plusieurs files d'attentes

Les moniteurs : sémantique

- **Seul un processus (ou tâche ou thread) peut être actif à un moment donné à l'intérieur du moniteur**

- **La demande d'entrée dans un moniteur (ou d'exécution d'une primitive du moniteur) sera bloquante tant qu'il y aura un processus actif à l'intérieur du moniteur**

-> L'accès à un moniteur construit donc implicitement une exclusion mutuelle

Les moniteurs : sémantique (suite)

- **Lorsqu'un processus actif au sein d'un moniteur ne peut progresser dans son travail (une certaine condition est fausse, ou lorsqu'il lui manque certaines ressources) il faut pouvoir "rendre" l'accès au moniteur et mettre « en attente » le processus actif**

- **De même il faudra pouvoir « réveiller » un processus en attente lorsque l'on modifie les variables internes du moniteur**

- **Il existe pour cela deux types de primitives**
  - wait : qui met en attente l'appelant et libère l'accès au moniteur
  - signal : qui réveille un des processus en attente à l'intérieur du moniteur (un processus qui a exécuté précédemment un wait)

Les moniteurs : sémantique (suite)

- **Selon les langages (ou les normes) ces mécanismes peuvent être implémentés de différentes façons**
  - méthodes « wait / notify / notifyAll » en Java et méthodes « synchronized »
  - primitives « pthread_cond_wait / pthread_cond_signal » en Posix et variables conditionnelles
  - gardes associées aux entrées en Ada, instruction « requeue » et objets protégés
- **La sémantique des réveils peut varier :**
  - Qui réveille t-on (le plus ancien, le plus prioritaire, un choisi au hasard, …)
  - Quand réveille t-on (dès la sortie du moniteur, au prochain ordonnancement, …)

Les moniteurs : exemple

- **Dans l'exemple précédent il faut encapsuler le registre et les opérations de manipulations du registre dans un moniteur**

- **Définition du moniteur**

Données Internes :
valeur locale du registre (R_Local)

Points d'entrée :
Incrémente_Le_Registre
Lecture du registre

Primitives_Internes :
Ecriture du registre

Synchronisation en mémoire partagée : bilan

- **Sémaphores** :
  - mécanisme simple mais qui peut conduire à des erreurs subtiles
  - demande une grande rigueur dans leur utilisation
  - à réserver pour la mise en place de petite section critique

- **Moniteurs** :
  - mécanisme de plus haut niveau
  - simplifie la mise en place de section critique
  - peut néanmoins conduire à des erreurs (mauvaise protection, trop de points d'entrées)

Conclusion

- **En modèle réparti la communication et la synchronisation entre tâches se fait par l'échange de messages; on utilise principalement deux mécanismes :**
  - L' appel de procédures à distance (RPC), mécanisme plutôt système
  - L'invocation de méthodes distantes (RMI), mécanisme plutôt langage
- **En modèle centralisé la communication et la synchronisation reposent sur le partage de données communes; il faut alors « protéger » l'accès ces données à l'aide de deux mécanismes :**
  - Les sémaphores, plutôt système
  - Les moniteurs, plutôt langage